

# UNCLASSIFIED

<b>AD NUMBER</b>
ADB208308
<b>NEW LIMITATION CHANGE</b>
<b>TO</b> Approved for public release, distribution unlimited
<b>FROM</b> Distribution authorized to DoD only; Proprietary Info.; Feb 96. Other requests shall be referred to AFMC/STI, Kirtland AFB, NM 87117-5776.
<b>AUTHORITY</b>
Phillips Lab [AFMC], Kirtland AFB, NM ltr dtd 28 Jul 97

THIS PAGE IS UNCLASSIFIED

# GRAPHICAL USER INTERFACE (GUI) INDEPENDENT EXPERT SHELL

Dustin Huntington

EXSYS Inc.  
1720 Louisiana Blvd., Suite 312  
Albuquerque, NM 87110

February 1996

Final Report

Distribution authorized to DoD components only; Proprietary Information; February 1996. Other requests for this document shall be referred to AFMC/STL.

**WARNING** - This document contains technical data whose export is restricted by the Arms Export Control Act (Title 22, U.S.C., Sec 2751 et seq.) or The Export Administration Act of 1979, as amended (Title 50, U.S.C., App. 2401, et seq.). Violations of these export laws are subject to severe criminal penalties. Disseminate IAW the provisions of DoD Directive 5230.25 and AFI 61-204.

**DESTRUCTION NOTICE** - For classified documents, follow the procedures in DoD 5200.22-M, Industrial Security Manual, Section II-19 or DoD 5200.1-R, Information Security Program Regulation, Chapter IX. For unclassified, limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.



**PHILLIPS LABORATORY**  
Space and Missiles Technology Directorate  
AIR FORCE MATERIEL COMMAND  
KIRTLAND AIR FORCE BASE, NM 87117-5776

19960401 069

DTIC QUALITY INSPECTED 1

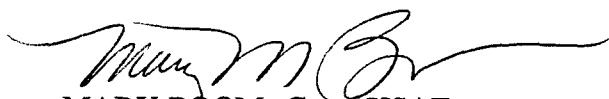
Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data, does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report contains proprietary information and shall not be either released outside the government, or used, duplicated or disclosed in whole or in part for manufacture or procurement, without the written permission of the contractor. This legend shall be marked on any reproduction hereof in whole or in part.

If you change your address, wish to be removed from this mailing list, or your organization no longer employs the addressee, please notify PL/VTX, 3550 Aberdeen Ave SE, Kirtland AFB, NM 87117-5776.


Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

This report has been approved for publication.




MARY BOOM, Capt, USAF  
Project Manager

FOR THE COMMANDER



CHRISTINE M. ANDERSON, GM-15  
Chief, Satellite Control and Simulation  
Division



HENRY L. PUGH, JR., Col, USAF  
Director, Space and Missiles Technology  
Directorate

# DRAFT SF 298

<b>1. Report Date (dd-mm-yy)</b> February 1996		<b>2. Report Type</b> Final		<b>3. Dates covered (from... to )</b> 04/95 to 12/95	
<b>4. Title &amp; subtitle</b> Graphical User Interface (GUI) Independent Expert Shell				<b>5a. Contract or Grant #</b> F29601-95-C-0092	
				<b>5b. Program Element #</b> 65502F	
<b>6. Author(s)</b> Dustin Huntington				<b>5c. Project #</b> 3005	
				<b>5d. Task #</b> C0	
				<b>5e. Work Unit #</b> KP	
<b>7. Performing Organization Name &amp; Address</b> EXSYS Inc. 1720 Louisiana Blvd., Suite 312 Albuquerque, NM 87110				<b>8. Performing Organization Report #</b>	
<b>9. Sponsoring/Monitoring Agency Name &amp; Address</b> Phillips Laboratory 3550 Aberdeen Ave SE Kirtland AFB, NM 87117-5776				<b>10. Monitor Acronym</b>	
				<b>11. Monitor Report #</b> PL-TR-96-1020	
<b>12. Distribution/Availability Statement</b> Distribution authorized to DoD components only; Proprietary Information; February 1996. Other requests for this document shall be referred to AFMC/STI.					
<b>13. Supplementary Notes</b>					
<b>14. Abstract</b> The objective of this effort was to demonstrate the feasibility of creating an expert system development and delivery environment that would allow expert systems to be developed using easy to learn shells to produce systems which can be integrated in a Graphical User Interface (GUI) independent manner within a wide range of user interfaces and on a wide range of platforms. This effort successfully partitioned an existing commercial expert system program, separating the interface engine from the GUI. The ability of the expert systems to be queried and provide information about decisions and the decision making process was retained. This was accomplished by designing data structures and protocols that allow the expert system to function and to interface with other programs. Direct integration was demonstrated in this proof of concept, as was the feasibility of converting the runtime program into a form that could be accessed by multiple programs simultaneously. This report details the proof of concept only.					
<b>15. Subject Terms</b> Rule-based systems, Artificial Intelligence, Expert systems					
<b>Security Classification of</b>			<b>19. Limitation of Abstract</b>	<b>20. # of Pages</b>	<b>21. Responsible Person (Name and Telephone #)</b>
<b>16. Report unclassified</b>	<b>17. Abstract unclassified</b>	<b>18. This Page unclassified</b>	Limited	94	Capt Mary Boom (505) 846-0461 x317

**GOVERNMENT PURPOSE LICENSE RIGHTS  
(SBIR PROGRAM)**

**Contract Number: F29601-95-C-0092**

**Contractor: EXSYS Inc.**

**Albuquerque, NM 87110**

For a period of four (4) years after delivery and acceptance of the last deliverable item under the above contract, this technical data shall be subject to the restrictions contained in the definition of "Limited Rights" in DFARS clause at 252.227-7013. After the four-year period, the data shall be subject to the restrictions contained in the definition of "Government Purpose License Rights" in DFARS clause at 252.227-7013. The Government assumes no liability for unauthorized use or disclosure by others. This legend shall be included on any reproduction thereof and shall be honored only as long as the data continues to meet the definition on Government purpose license rights.

The software and programs accompanying this report were developed as a proof-of-principle of the concept of separating the inference engine from the user interface. They are NOT designed or intended to be used as a finished or commercial expert system runtime program. The software is provided as is to demonstrate the ability to use different user interfaces. No other warranties or guarantees of suitability or usefulness are made.

# TABLE OF CONTENTS

	Page
<b>1. OBJECTIVES .....</b>	<b>1</b>
<b>2. IDENTIFICATION AND SIGNIFICANCE OF THE PROBLEM .....</b>	<b>2</b>
<b>3. PHASE 1 TECHNICAL OBJECTIVES .....</b>	<b>4</b>
3.1 TASK 1 - API PROTOCOL DESIGN .....	4
3.2 TASK 2 - CORE INFERENCE ENGINE REWRITE. ....	5
3.3 TASK 3 - DIRECT INTEGRATION .....	5
3.4 TASK 4 - DLL PROOF OF CONCEPT .....	5
<b>4. RESULTS .....</b>	<b>6</b>
4.1 TASK 1 - API PROTOCOL DESIGN .....	6
<i>Ask for a Qualifier Value</i> .....	7
<i>Ask for a Variable Value.</i> ....	8
<i>Displaying a Rule</i> .....	9
<i>Displaying Results</i> .....	12
4.2 TASK 2 - CORE INFERENCE ENGINE REWRITE .....	15
4.3 TASK 3 - DIRECT INTEGRATION .....	16
4.4 TASK 4 - DLL PROOF OF CONCEPT .....	21
<b>5. ADDING CUSTOM COMMANDS .....</b>	<b>22</b>
5.1 EX_LINK.C .....	23
5.2 NAMING CUSTOM COMMANDS .....	24
5.3 BODY OF THE CUSTOM COMMAND .....	25
5.4 PASSING ARGUMENTS TO CUSTOM COMMANDS.....	26
5.5 SETTING VALUES IN CUSTOM COMMANDS .....	27
<b>6. EXSYS API FUNCTIONS .....</b>	<b>29</b>
6.1 FUNCTIONS .....	30

# 1. Objectives

---

Expert systems are widely used in the Air Force and have a proven to be an excellent solution in a wide range of problems (e.g. diagnostic, fault isolation, preventive maintenance, help desks, repair assistance). However, to be implemented widely and effectively, expert systems need to be developed in a cost effective manner and integrated within existing programs through the user interfaces currently in place. Current expert system tools lack a generic approach to integration of developed applications into existing programs. Applications which have been tightly integrated have typically used a highly customized approach that both made the development of the expert system logic costly and resulted in such customized code that it could not be easily applied to other applications. This project was to demonstrate the feasibility of creating an expert system development and delivery environment that would allow expert systems to be developed using easy to learn shells to produce systems which can be integrated in a "graphical user interface (GUI)" independent manner within a wide range of user interfaces and on a wide range of platforms.



## 2. Identification and Significance of the Problem

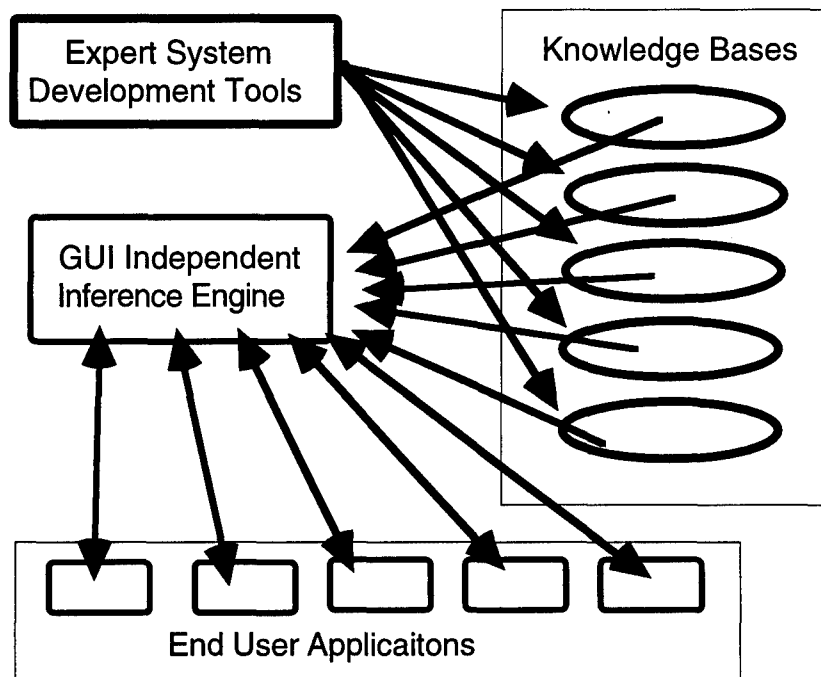
---

Expert systems have been used to solve a wide range of problems in the military. In most cases these were custom developed applications with unique end user interfaces. Development was often custom programmed in an "AI language", such as LISP or Prolog. These system have often been highly successful and effective in solving the problem, but the cost per system has been very high. Also, there have been many difficulties in integrating the expert system with existing software and user interfaces. These problems have been due to limitations in the language or shell used to develop expert system.

To implement expert systems widely and in a cost effective way requires:

1. An expert system development environment (shell) which facilitates development of applications rapidly and at low cost. Ideally this is achieved by allowing the staff with domain expertise build the applications themselves.
2. A highly flexible runtime/system delivery environment that enables the expert system decision making capability to be integrated into a wide range of existing applications. This can be used to both enhance the ability of existing programs to solve problems while retaining the application interface users are familiar with. Also, new programs can be designed to meet user needs without being limited by the interface capabilities of expert system tools. This delivery system should be independent of the GUI used. It should function in a client/server mode to provide knowledge (answers) to the client's input of relevant data.

The proposed tools would meet these requirements. Expert system knowledge bases would be developed with existing development tools. The EXSYS development tools have a well proven track record of allowing rapid development of complex expert system applications. The knowledge bases created could be run from a GUI independent inference engine that could be easily embedded within a variety of applications and accessed by multiple users.



### **3. Phase 1 Technical Objectives**

---

The Phase 1 technical objectives were to take the existing EXSYS Expert System Development Tools and enhance them to function in a GUI independent mode. With thousands of users the EXSYS development tools are well proven to make expert system development easy and rapid. By starting with the proven current EXSYS tools, a tremendous amount of work was saved in the creation of the development tools. The main effort focused on the delivery system by producing a GUI independent inference engine that can run applications developed with the existing EXSYS tools.

The display of data to end users has become increasingly sophisticated and specialized. A wide range of programs have been developed that allow highly customized (or standardized) displays of data. While such programs have excellent ability to display information, they lack an expert system inference engine. The tight integration of the inference engine with specialized data display programs usually requires custom programming to bypass the default user interface provided by the expert system. This makes the integration difficult and substantially increases the cost and time to build applications.

The Phase 1 technical objective was to overcome these limitations through a re-design of the EXSYS runtime to make it:

1. GUI independent, so that the EXSYS functions do not limit the end user application in its user interface, while still retaining the full functionality of the expert system to interactively explain the reasoning used to the end user--one of the most powerful parts of an expert system.
2. Design the GUI independent version so that multiple queries can be effectively handled allowing future development of a client/server architecture.

#### **3.1 Task 1 - Application Program Interface (API) Protocol Design**

The first task was to design a set of API protocols and data structures required for the expert system to function and interface with the other programs. These protocols include ways to access the expert system in a GUI independent way while retaining the full ability to

interrogate the expert system to request an explanation of the logic used to reach conclusions. Protocols to handle the full functionality of the applications developed with the EXSYS expert system development tools were developed. These protocols were designed in such a way that both GUI and non-GUI programs are able to access the EXSYS functions.

### **3.2 Task 2 - Core Inference Engine Rewrite**

The core inference engine of the EXSYS Runtime program was rewritten to reflect the new protocols. This allowed a GUI independent inference engine to be developed. This core engine does not have any direct calls to a native operating system GUI command or any intermediate GUI interface layer (XVT).

### **3.3 Task 3 - Direct Integration**

The GUI independent version of the API was used as the core for a custom interface program. A user interface was wrapped around the core program using a third party "GUI design" program. This demonstrates the functionality of the core inference engine and its ability to be integrated with other programs. Wrap around layers allowing normal Runtime functionality were designed both using the XVT libraries and a character only layer. This demonstrates the feasibility to run/test applications with a stand alone runtime that will function identically to the GUI independent core Inference Engine.

### **3.4 Task 4 - DLL Proof of Concept**

The GUI independent version of the EXSYS Runtime was designed in such a way that it could be converted to a DLL, allowing multiple applications to access the inference engine simultaneously. While a full development of this approach was not possible in Phase 1, the feasibility and effectiveness of this concept was tested.

## 4. Results

---

### 4.1 Task 1 - API Protocol Design

The goal of phase I was to convert the current EXSYS program from a form where the code for the GUI screens was intermingled with the code for the inference engine, to one where the inference engine could be separated from the user interface. This required major changes in the user interface routines. The existing version of EXSYS used end user interface routines that extensively accessed the EXSYS data structures for information needed to create the user display screen. The approach demonstrated in this project was to create data structures that could be passed to the display routines (either GUI or non-GUI) where the data structure would contain all of the information needed to display the screen. In this way, the display routine does not further direct rapid access to the EXSYS data structures. The routine displaying the data, or asking the question, could even be in a separate executable connected through DDE. The first goal to achieve this was design of data structures to:

1. Ask for an EXSYS Qualifier (multiple choice question) value.
2. Ask for the value of a numeric variable.
3. Display a rule.
4. Display results.
5. Display error or message text.

Each data structure had to be:

1. fully self contained and provide all data/text needed to create the appropriate screen.
2. Flexible to handle different amounts of data.
3. Of a suitable type to be passed via DDE (no pointers to other data).

These data structures are passed to a routine that will ask for data or present information.

Each data structure will be discussed individually:

### **Ask for a Qualifier Value**

```
#define ASK_QUAL_PROMPT_LTH  500
#define ASK_QUAL_VALUE_LTH  500

typedef struct ask_qual_struct {
    int qualnum;                /* number of the qualifier */
    char prompt[ASK_QUAL_PROMPT_LTH]; /* prompt text */
    int numval;                 /* number of values */
    int maxval;                 /* maximum values */
    char value[31][ASK_QUAL_VALUE_LTH]; /* value text - first is value[1] */
    int num_selectable;         /* number of values that can be selected */
    int allow_why;              /* allow WHY 1=allow 0=no */
    int start_ask;              /* 1=data asked due to .cfg or .cmd file, else =0 */
    char retbuff[200];
    long custhelp;
    long custscr;
    long windata;
};
```

**qualnum** - The number of the qualifier. Used in queries for additional information.

**prompt** - The text of the qualifier prompt.

**numval** - The total number of values to choose among.

**maxval** - The maximum number of separate values that the end user can select. This is usually either a single value (1) or all values.

**value[31]** - The text of the individual values that the end user will select among. The first is value[1].

**num\_selectable** - Currently redundant with "maxval". Included for future expansion.

**allow\_why** - Flag to indicate if the user is allowed to ask "WHY" in response to the question. This is dependent on where the question was asked from within EXSYS. A value of 1=allow WHY, a value of 0 means WHY is not permitted.

**start\_ask** - Flag to indicate what EXSYS function lead to the question being asked. 1=data asked due to .cfg or .cmd file, else=0.

**retbuff** - The buffer to write the return data into. The format of the data is the same as a return.dat line. This data is used to set the values in the program when the routine using the data structure returns.

**custhelp** - The offset for the custom help commands in the <kb>.hlp file.

**custscr** - The offset for the custom screen commands in the <kb>.scr file.

**windata** - A long int that can be used by the developer to associate any other data with the structure. This may be a pointer cast as a long int pointing to any other structure, e.g. window handles on the display window. The use of this element is entirely up to the end developer.

### Ask for a Variable Value

```
#define ASK_VAR_PROMPT_LTH 500

typedef struct ask_var_struct {
    int varnum; /* number of the variable */
    char prompt[ASK_VAR_PROMPT_LTH]; /* prompt text */
    char var_type; /* type string='s' numeric='n' */
    char limits;
    /* flag that there are limits 0=no 1=yes */
    double upperlim; /* upper limit value */
    double lowerlim; /* lower limit value */
    char retbuff[300];
    long custhelp;
    long custscr;
    long windata;
};
```

**varnum** - The number of the variable. Used in queries for additional information.

**prompt** - The text of the variable prompt.

**var\_type** - The type of data expected. 'S'=string data, 'N'=numeric data (the input string will be converted to a float with atof() ).

**limits** - Are there limits specified for the acceptable range on input values. (Applies only to numeric variables) 1=limits, 0=no limits.

**upperlim** - The upper limit value. If limits are specified, the input value must be less than or equal to this value.

**lowerlim** - The lower limit value. If limits are specified, the input value must be greater than or equal to this value.

**retbuff** - The buffer to write the return data into. The format of the data is the same as a return.dat line. This data is used to set the values in the program when the routine using the data structure returns.

**custhelp** - The offset for the custom help commands in the <kb>.hlp file.

**custscr** - The offset for the custom screen commands in the <kb>.scr file.

**windata** - A long int that can be used by the developer to associate any other data with the structure. This may be a pointer cast as a long int pointing to any other structure, e.g. window handles on the display window. The use of this element is entirely up to the end developer.

### **Displaying a Rule**

Displaying a rule in an efficient manner with a data structure that could be passed over DDE presented special problems. Rules can vary greatly in size from 2 conditions to 384 conditions. Also the amount of text in a specific condition can vary greatly. To be able to pass the structure over DDE, the structure could not simply use pointers to other data items that would allow the pieces of rule text to be created. All text required had to be self contained in a fixed size structure. To allocate enough space for each of the possible conditions in an array of the conditions would have resulted in very large (384 \* 500) char byte structures, which would have still limited the length of an individual condition to 500 char. Instead a 10000



char buffer was used to hold all of the text. This buffer is passed as part of the structure, and the individual conditions simply contain offsets into the structure to read text. Individual text item can be any length, provided collectively they are less than 10000 bytes. The individual strings are recovered with a API command:

```
get_rule_text(offset,      /* the offset in the file */
              struct disprule_struct, /* the rule struct */
              char *retbuffer) /* the return text buffer */
```

The use of `get_rule_text()` is demonstrated in the example of displaying a rule.

```
#define MAX_RULE_TEXT 10000

typedef struct disprule_struct {
    int rulenum;
    char name[18];
    int used;
    int ifnum;
    struct {
        long text;
        int status;
        char type;
        /* Q=qualifier V=variable/formula C=choice */
        int item_num; /* item number */
        union
        {
            {long value;
             struct
             {int type;
              int formnum;
              } ch;
             } val;
        } ifcond[128];
        int thennum;
        struct {
            long text;
        } thencond[128];
        int elsenum;
        struct {
            long text;
        } elsecond[128];
        long note;
        long ref;
        int winnum;
        long windata;
        int toprule;
        char textstr[MAX_RULE_TEXT];
    };
};
```

**rulenum** - The number of the rule.

**name** - The name of the rule.

**used** - A flag to indicate if the rule has fired. 1=fired TRUE,  
-1= fired FALSE, 0=unknown (Not yet fired).

**ifnum** - The number of IF conditions in the rule.

**ifcond.text** - The offset of the text of the IF of a specific IF condition.  
The text is recovered by calling `get_rule_text()`.

**ifcond.status** - The status of the specific IF condition 1=TRUE,  
-1= FALSE, 0=unknown (No data yet).

**ifcond.type** - The type of item in the condition. Q=qualifier,  
V=variable/formula, C=choice.

**ifcond.item\_num** - The number of the specific item.

**ifcond.val** - Information for requesting additional data. Not  
implemented under Phase 1, and not required for simply displaying  
the rule.

**thennum** - The number of THEN conditions in the rule.

**thencond.text** - The offset of the text of the THEN of a specific THEN  
condition. The text is recovered by calling `get_rule_text()`.

**elsenum** - The number of ELSE conditions in the rule.

**elsecond.text** - The offset of the text of the ELSE of a specific ELSE  
condition. The text is recovered by calling `get_rule_text()`.

**note** - The offset for the note text associated with the rule. If this is a  
value of 0, there is no note text. The text is recovered with  
`get_rule_text()`.

**ref** - The offset for the reference text associated with the rule. If this  
is a value of 0, there is no reference text. The text is recovered with  
`get_rule_text()`.

**winnum** - The number of the window associated with the rule. This  
value can be set and used by the end developer as needed.

**windata** - A long int that can be used by the developer to associate  
any other data with the structure. This may be a pointer cast as a  
long int pointing to any other structure, e.g. window handles on the

display window. The use of this element is entirely up to the end developer.

**toprule** - The maximum rule number in the system. This is needed for user interfaces that allow the end user to select to display another rule from the rule display window. It is necessary to know the maximum legal value to prevent asking for rule numbers not in the system.

**textstr** - The block of text containing all text string used to build the rule. This element should not be accessed directly. All text in the block can be recovered by the `get_rule_text` command.

### Displaying Results

The display of results presented the same problems as the display of a rule - the text required can vary from a few short strings to many long strings. This needed to be handled efficiently and in a way that could be passed via DDE. The solution was the same as used for rule display. The text strings are put into a large block of text that is passed as part of the structure. The text in the block is accessed via a command:

```
get_result_text(offset,      /* the offset in the file */
                 struct nogui_result_struct, /* the struct */
                 char *retbuffer) /* the return text buffer */
```

The use of `get_result_text()` is demonstrated in the example of displaying a rule.

The result structure is used for 4 different types of screens in EXSYS:

1. Display of the final or intermediate results.
2. Display of current choice values.
3. Display of current qualifier/variable values.
4. Change and Rerun.

```
#define MAX_ITEM_NUM 500
#define MAX_RESULT_TEXT 10000
```

```
typedef struct nogui_result_struct {
    int disp_type;
    /* 0=results 1=choice 2=qual/var 3=change */
    double disp_thr; /* threshold for display */
```

```

int dispall;          /* display all values */
int sorted;           /* already sorted TRUE / FALSE */
int topch;            /* highest item assigned data */
char rptfile[100];    /* report file to use */
int runagain;
    /* allow another run=1 or not=0 (cmd language) */
long windata;         /* user window data */
struct item_struct {
    int num;           /* number of item */
    char type;         /* type of item C, Q, V */
    char vartype;      /* type of item N, S, T */
    long text;
    char asked;        /* flag value was asked */
    char calc;         /* flag for value calculated */
    int disp;          /* flag for display at end 0=no +-1=yes */
    char init;         /* flag to initialize variable */
    union {
        double val;   /* numeric value */
        long sval_offset; /* string offset */
        long qval;    /* qualifier value */
    }v;
    }item[MAX_ITEM_NUM];
char textstr[MAX_RESULT_TEXT];
struct choicevalue chval[];
};

```

**disp\_type** - Type of screen to display. 0=results, 1=choice, 2=qual/var, 3=change/rerun.

**disp\_thr** - Lowest confidence value threshold. Choices with final confidence below this threshold should not be displayed.

**dispall** - Flag to display all values regardless of final value - even those which did not receive a value. 1=Display all, 0=no.

**sorted** - Flag to indicate if the data has been sorted according to value or not. 1=sorted, 0=not sorted. (Normally this will be sorted).

**topch** - Highest item array value that contains data.

**rptfile** - Name of a report file to run as an option from the results window.

**runagain** - Flag to allow a "Run Again" option. 1=Allow, 0=Not allowed.

**windata** - A long int that can be used by the developer to associate any other data with the structure. This may be a pointer cast as a long int pointing to any other structure, e.g. window handles on the display window. The use of this element is entirely up to the end developer.

Note: item.xxx elements other than "item.text" are not normally directly needed by display routines. They are used in sorting routines and to display additional information about an item.

**item.num** - The number of a data element in the results list.

**item.type** - The type of a data element in the results list.  
'C'=Choice, 'Q'=Qualifier, 'V'=Variable.

**item.vartype** - For Variables, the type of variable. 'N'=Numeric, 'S'=String, 'T'=Text only (Message).

**item.text** - The offset for the text of the data element in the results list. The actual text is recovered with get\_result\_text() using this offset.

**item.asked** - Flag that the data was asked of the end user.  
1=Asked, 0=not asked.

**item.calc** - Flag that the data was calculated or derived by the inference engine. 0=Not calculated, 1=partially calculated (Inference engine not full complete), 2=Fully calculated.

**item.disp** - Flag to display item with the results. 1=Yes, 0=No.

**item.init** - Flag that the variable was originally initialized.

**item.v.val** - Value for numeric variables or choices.

**item.v.sval\_offset** - Offset for value of string variables. The actual text is recovered with get\_result\_text() using this offset.

**item.v.qval** - Value for qualifiers.

**textstr** - The block of text containing all text string used to build the display screen. This element should not be accessed directly. All text in the block can be recovered by the get\_display\_text command.

**chval** - An array used in sorting the choices. This array should not be directly accessed by the display routine.

## 4.2 Task 2 - Core Inference Engine Rewrite

The core inference engine of EXSYS was rewritten to make it GUI independent. The test of this was the ability to compile the core inference engine routine without any of XVT include files and to link a finished executable without using the XVT libraries. All communication with the routines to ask questions, display a rule and display results were done through the data structures described in 4.1.

The GUI independent core inference engine routines provided with this report are:

```
exnogui0.obj  
exnogui1.obj  
exnogui2.obj  
exnogui3.obj  
exnogui4.obj  
exnogui5.obj  
exnogui6.obj  
exnogui7.obj  
excom.obj  
exread.obj  
excust.obj  
ex_link.obj  
ex_link3.obj
```

In addition, there are a set of GUI independent routines that handle the EXSYS interface to other programs. For this proof of concept, the NULL version of the graphics, spreadsheet, PI and SQL interfaces was used. These routines resolve the references in the link files, but are not functional. The external commands for table, frame and blackboard commands are functional. (Note: the graphics and SQL routines are intrinsically based on MS Windows and the functional versions can not be put in a non-Windows program.)

```
eximnull.obj  
exssnull.obj  
exqenull.obj  
expinull.obj  
exsysbb.obj
```

These core inference engine routines can be linked with either a GUI or non-GUI user interface.

### 4.3 Task 3 - Direct Integration

The core inference engine routines were linked with both GUI and non-GUI wrap-arounds to produce executable runtime versions of EXSYS capable of reading and executing knowledge bases developed with the off-the-shelf version of the EXSYS editor.

The wrap around .OBJ files are included with this report.

To build a non-GUI version of the EXSYS Runtime, link the .OBJ modules listed in 4.2 with the module:

```
exguifct.obj  
ex_link2.obj
```

(A RTLINK file to build the character version is in ex\_char.lnk)

To build the GUI (XVT) version of the EXSYS Runtime, link the .OBJ modules listed in 4.2 with the modules:

```
exsysp0.obj  
exsysp1.obj  
exsyspla.obj  
exsysp3.obj  
exsysp4.obj  
exsysp5.obj  
exsysp6.obj  
exsysp7.obj  
exsysp8.obj  
exsysp9.obj  
exsysp11.obj  
exsysp12.obj  
exsysp13.obj  
exsysp14.obj  
guilink2.obj
```

(The nmake file to build the XVT version is in the file exsysp.mk)

#### Example:

In both cases, the inference engine code is the same. Custom user routines can be added to handle the end user interface. In the case of the XVT code, these routines are fairly long and complex, so the character routines are used to demonstrate the API for the end user interface.

The character version in `ex_char.c` was designed to use the simplest possible user interface routines. Only `printf()` and `scanf()` functions were used for I/O. This means that the character version could even be used on a computer with only paper printer output and character input. (The target simplest machine was taken to be comparable to the old "Silent 700" terminals - the quintessential non-GUI machine.)

### Asking Questions:

To ask for a qualifier input the routine `win_ask()` is used. It is passed a pointer to a structure `ask_qual_struct` which has all of the information needed to ask the question. Input is returned by copying the input to the `retbuff` element in the structure.

```
win_ask(struct ask_qual_struct *askdatap)
{
    int i;
    char b[100];

startloop:

    printf("\n\n\n-----\n\n");

    if (askdatap->maxval != 0)
        printf(" SELECT UP TO %d\n", askdatap->maxval);
    else
        printf("SELECT:\n");

    printf("%s\n", askdatap->prompt);

    for (i=1; i<=askdatap->numval; i++)
        printf("  %d  %s\n", i, askdatap->value[i]);

    scanf("%s", b);

    if (strlen(b) > 0)
        strcpy(askdatap->retbuff, b);
    else
        goto startloop;
}
```

This routine first prints a line and uses the `maxval` element of the structure to tell the user how many item may be selected. The `prompt` element is the text of the qualifier. Each of the `value[ ]` elements is one of the values. The total number of values is in the `numval` element. When data is returned from the `scanf()` it is copied to the `retbuff` element and the function returns. When the `win_ask()` routine returns, the text in the `retbuff` element is automatically used to set the values for the qualifier.



Clearly this routine is about as simple as possible. Many types of formatting could be added. In a GUI environment, while the routine would have to be event driven, it would have the same basic functions - display the values and accept input.

To ask for a variable value, the routine is even simpler:

```
win_askvar(struct ask_var_struct *ask_var_datap)
{
    char b[100];

    startloop:

    printf("\n\n\n-----\n\n");
    printf(" INPUT A VALUE:\n");
    printf("%s\n", ask_var_datap->prompt);

    scanf("%s", b);
    if (strlen(b) > 0)
        strcpy(ask_var_datap->retbuff, b);
    else
        goto startloop;
}
```

In a full implementation, there would be formatting and probably type checking to separate string and numeric values.

If the end user enters "WHY" in response to a question, the rule(s) being tested will be displayed. This is handled automatically by the code that processes the retbuff element after win\_ask() and win\_askvar() return.

The display of a rule is a little more complicated because the text needed for portions of the rule is stored in a single block, rather than as easily accessible strings. The add\_disp\_rule() routine is called and passed a pointer to the structure disprule\_struct. In addition two flags are passed:

1. A how\_flag indicating if HOW can be used.
2. A wait flag indicating if the called routine should wait while the rule is displayed or continue processing. Note that the wait flag only has significance in event driven (GUI) systems where the program can display a window and wait for it to be dismissed, or display a window and continue other processing.

```

add_disp_rule(struct disprule_struct *rule,
               int how_flag,
               int wait)
{
    int i, k;
    char b[500];

    printf("\n\nRule %d\n", rule->rulenum);
    printf("\nIF:\n");
    for (i=1;i<=rule->ifnum;i++)
        {get_rule_text(rule->ifcond[i-1].text, rule, b);
         printf("%d  %s\n", i, b);
        }
    printf("\nTHEN:\n");
    for (i=1;i<=rule->thennum;i++)
        {get_rule_text(rule->thencond[i-1].text, rule, b);
         printf("    %s\n", b);
        }
    if (rule->elsenum > 0)
        {
            printf("\nELSE:\n");
            for (i=1;i<=rule->elsenum;i++)
                {get_rule_text(rule->elsecond[i-1].text, rule, b);
                 printf("    %s\n", b);
                }
        }

    askagainloop:

    printf("\nOK?: ");

    i=0;

    while ((k=getch()) > 13)
        {b[i] = k;
         i++;
         printf("%c", k);
        }
    b[i] = 0x00;

    i = atoi(b);

    if (i > 0)
        {nogui_findsource(FALSE, FALSE,
                          rule->ifcond[i-1].type,
                          rule->ifcond[i-1].item_num,
                          0, 0, 0);
         goto askagainloop;
        }
}

```

Each of the IF, THEN and ELSE conditions text strings are created with the routine `get_rule_text()` which returns the string identified by the `ifcond[]`, `thencond[]` and `elsecond[]` elements. Once the rule is displayed, the end user may ask the source of any of the IF conditions by entering their number. The routine `nogui_findsource()` is then called to display the source for the particular piece of information.

Results are displayed by simply displaying the elements of the passed structure `nogui_result_struct`. Individual items of text are recovered with the `get_result_text()` routine.

```
dispresults(struct nogui_result_struct *res)
{
    int i;
    char b[300];

    printf("\n\nResults:\n");

    for (i=1; i<=res->topch; i++)
    {
        get_result_text(res->item[i].text, res, b);
        if (res->item[i].type == 'C')
            printf("%d - %s : %lf \n", i, b,
                res->item[i].v.val);
        else if (res->item[i].type == 'Q')
            printf("%d - %s \n", i, b);
        else if (res->item[i].type == 'V')
            printf("%d - %s = %lf \n", i, b,
                res->item[i].v.val);
    }
}
```

These 4 sample routines described in the above show a simple user interface can be added onto the GUI independent inference engine. Since the inference engine does not do any event processing itself, it can be called as a routine from either a GUI or non-GUI program.

### Examples on Disk

The disk contains examples of both the GUI and non-GUI versions of the EXSYS Runtime built with the GUI Independent Linkable Objects. The CHAR subdirectory contains the program EXSYSP.EXE - This is the character version. It must be passed the name of the file to run. For example, EXSYSP ZZZ where ZZZ is the name of the knowledge base to run. This program must be run from a DOS window. As text is displayed, it just scrolls up the window.

The GUI subdirectory also contains a different EXSYSP.EXE which is the GUI version built with the same linkable objects. It behaves the same as a normal GUI version of the EXSYS Runtime. It may be run from MS Windows or Windows NT.

The object modules and make files used to build these examples are in the OBJ directory.

## **4.4 Task 4 - DLL Proof of Concept**

The possibility of converting the .OBJ files into a DLL format was examined. While it was determined that such an approach was possible, there were limitations. These were due to the requirements for separate data segments in a DLL and the calling program. Phase 1 was designed to be a proof of principle demonstration. Functionality for asking questions, display of rules and display of results was implemented in a GUI independent form that could be linked with various wrap-around layers. However, the proof of principle was not designed to fully separate the functionality at a level that would allow separate data segments. This would be possible, but was not implemented as part of Phase 1.

Simple routines were separated out and compiled to be in a DLL form that could be linked with the other program. Large sections of the GUI independent interface could be handled in this way. This was a proposed goal for the Phase 2 of this project.

In the current version, since the .obj files are linked directly, there would be no functional advantage to converting to a DLL form for some functions.

## 5. Adding Custom Commands

---

The easiest way to customize EXSYS programs is to add custom commands. These may be commands which:

- ♦ interface to special software or hardware.
- ♦ add special user interfaces (such as multimedia).
- ♦ access EXSYS API commands in a way that allows them to be called in the command language or rules.
- ♦ perform calculations that are not practical to do within EXSYS such as statistical functions.
- ♦ provide a way to access user defined data structures.

Anything that can be programmed in the C language can be added as a custom function. EXSYS Linkable Object Modules (Linkables) is the EXSYS inference engine in the form of .obj modules with C source code for up to 99 user defined commands.

## 5.1 EX\_LINK.C

Custom commands are added by modifying the code in the module `ex_link.c`. This module is provided as C code with Linkables. The distribution version of `ex_link.c` has stub functions for 99 commands.

The custom commands become EXSYS Internal Commands and can be called anywhere that other internal commands such as `RUN( )` can be called:

1. From the THEN / ELSE part of rules.
2. In the command file.
3. Associated with a qualifier or variable.

## 5.2 Naming Custom Commands

Each custom command must have a name that will be used in the rules and command files to call to the command. When the command is called, there are three parts:

1. All custom commands start with **CC-**
2. The name associated the command
3. The arguments (or parameters) for the command in ( ). If there are no arguments, just ( ).

For example, if there is a custom command named ARGTEST, it would be called from the rules by `CC-ARGTEST(...)`.

To add a custom command, start by making a copy of the file `ex_link.c` provided with EXSYS Linkable Objects. All custom command definitions are made in `ex_link.c` which can then be linked into both a custom Runtime and a custom Editor.

Open `ex_link.c` with a text editor. Look for the array of custom command names:

```
char *cust_cmd[101] = {  
    "",          /* name for cmd0 */  
    "",          /* name for cmd1 */  
    "",          /* name for cmd2 */  
    "",          /* name for cmd3 */  
    "",          /* name for cmd4 */  
    "",          /* name for cmd5 */  
}
```

Each name is associated with a stub function `cmd##`. Replace the "" with command names. Enter names that will be easy to remember. The names can be any number of letters, but must NOT include spaces. The commands do not have to be entered sequentially. Do NOT enter the CC-, EXSYS will add that automatically.

For example, to name a custom command ARGTEST:

```
char *cust_cmd[101] = {  
    "ARGTEST",   /* name for cmd0 */  
    "",          /* name for cmd1 */  
    "",          /* name for cmd2 */  
}
```

This would be called from the rules with `CC-ARGTEST(...)`.

## 5.3 Body of the Custom Command

Each custom command name has an associated stub function associated with it. The function is listed in the note to the right of the name. In the ARGTEST example above, the function is cmd0( ).

Look in the ex\_link.c file after the array of names. There are 100 stub function named cmd0( ) through cmd99( ).

```
cmd0(argc, argv)
    int argc;
    char **argv;
{}
```

```
cmd1(argc, argv)
    int argc;
    char **argv;
{}
```

```
cmd2(argc, argv)
    int argc;
    char **argv;
{}
```

The C code to be executed for the custom command is entered in the corresponding cmd## function. In the case of ARGTEST, this would be cmd0( ). The next custom command would be associated with cmd1( ), etc.

To continue the example, cmd0( ) will be modified to echo back the arguments passed to it.

```
cmd0(argc, argv)
    int argc;
    char **argv;
{
    int i;

    /* This function echoes the arguments that */
    /* are passed to the function using the */
    /* xvt_note( ) command */

    xvt_note("\nCC-ARGTEST called with argc = %d\n",
             argc);
    for (i=0; i<argc; i++)
        xvt_note("    argv[%d] = %s \n",i,argv[i]);
}
```



The `xvt_note( )` command will display a dialog box with the specified string in it. The string is defined with most of the same options as a `printf( )` command.

## 5.4 Passing Arguments to Custom Commands

The arguments that are passed to the custom commands are similar to the arguments passed to `main( )` in a standard C program, a count of the arguments, and an argument vector. If you are not familiar with argument vectors, please review them in a C language programming book.

The argument vector is parsed from the string in `( )` when the custom command is used in the expert system. The string in the command is broken into a series of argument vectors. Each word separated by a space is made into a separate argument in the vector, unless it is surrounded by `" "`. This is exactly the same as the way a command entered at the DOS prompt is broken up and passed to `main( )`. However, in addition, any EXSYS variables, or `[ ]` expressions will be evaluated before being passed to the custom command.

For example, suppose we use the custom command called `ARGVECT` and have an EXSYS variable `[X]` with a value of 7. In our expert system we use the custom command:

```
CC-ARGVECT(AAA "bbb ccc" ddd [x])
```

This will be passed to the custom command `cmd0( )`:

```
cmd0(argc, argv)
    int argc;
    char **argv;
```

In this case `argc` will be 4 - the total number of argument vectors:

<code>argv[0] = AAA</code>	The first word up to a space
<code>argv[1] = bbb ccc</code>	The space is part of the string because of the <code>" "</code>
<code>argv[2] = ddd</code>	The third argument
<code>argv[3] = 7</code>	The value of <code>[x]</code> is evaluated before it is passed

The use of argument vectors for custom commands allows maximum flexibility in passing data to the custom command. There is no fixed number of arguments. Also, a single command can be made

polymorphic and used for multiple functions based on the number of arguments passed to it.

The demo system in the DEMO1 subdirectory contains the ARGTEST function. To test how argument vectors are passed, you can call this function with various combinations of parameters.

Functions that do not require any arguments can ignore the argc and argv values, but **MUST** include them in the definition of cmd##( ). The cmd##( ) functions must **ALWAYS** be defined as:

```
cmd#(argc, argv)
    int argc;
    char **argv;
    { ...
```

## 5.5 Setting Values in Custom Commands

The custom command functions do not return any value directly. However, the code associated with the function can use the EXSYS API functions to set values for EXSYS variables and qualifiers. Custom user data structures may also be used. Often it is necessary to pass an address to a custom function to tell it where to assign the value such as an EXSYS variable. This can be done using "".

For example, there is a custom command SETVAL that performs some operation and assigns the value to an EXSYS variable. This command could be called from the rules with:

```
CC-SETVAL(123, "[X] ")
```

The arguments in cmd#( ) would be:

```
argv[0] = 123
argv[1] = [X]
```

Note that argv[1] is the string "[X]", not the value of X. There are EXSYS API functions that will allow a value to be set for the EXSYS variable [X] using this string. For example:

```
noquote(argv[1]);
sprintf(b, "%s %lf", argv[1], value_to_assign);
set_data(b);
```

Other functions for setting values are discussed in the EXSYS API section in Chapter 6.

A common error is to call the command with:

```
CC-SETVAL(123, [X])
```

In this case, EXSYS will attempt to evaluate a value for [X] before it builds the argument vector and calls cmd#( ). This may cause other rules to be invoked and could be a source for infinite loops or other problems. When passing an identifier to an EXSYS variable or qualifier to a custom command either put it in quotes, or specify it in some way that will not be evaluated before the argument vector is built (e.g. CC-SETVAL(123, X))

## 6. EXSYS API Functions

---

There are a wide range of functions that can be called from custom commands or from your application to access EXSYS data structures or execute EXSYS functions. These are in addition to the API functions associated with the non-GUI embedding of the inference engine.

There are general categories of function which have been grouped together.

1. Functions to get data about various EXSYS data elements. These are typically used to build reports on the status of the data in the system or screens to ask questions.
2. Functions to set data. These are functions that allow data to be written directly into the EXSYS data structures.
3. Functions which provide access to high level EXSYS functions such as displaying a screen or running a command file.
4. Functions which allow intercepting events and allow the user to create custom versions of various EXSYS windows such as those used to display data or ask questions.
5. Utility routines to strip characters or provide conversions.

## 6.1 Functions

### **\*\* NOTE \*\***

Some API functions only make sense in a GUI environment. Those that can only be used in a GUI environment are listed in *italics*.

The EXSYS API functions should be used **ONLY** after `exsys_fileopen( )` has been called and the expert system started. Calling the functions prior to this point will result in a crash since the memory needed by the data structures will not have been allocated.

### **Functions to Get Data**

Text of a qualifier	<code>get_qual</code>
Text of a qualifier value	<code>get_qval</code>
Text of a qualifier and set values	<code>get_qv_set</code>
Text associated with a variable	<code>get_var</code>
Text of a choice	<code>choice_text</code>
Current value of a Choice	<code>get_choice_value</code>
Value of a numeric variable	<code>get_Nvar_value</code>
Value of a string variable	<code>get_Svar_value</code>
Test if a qualifier value is set	<code>get_qval_value</code>
Test if a qualifier was set	<code>qual_set</code>
Test if a variable was set	<code>var_set</code>
Determine type of a variable	<code>var_type</code>
List rules in backward chain	<code>rule_in_use</code>
Determine if a rule was used	<code>was_rule_used</code>

## Functions to Get Upper Limits

Number of choices	max_choice
Number of values for a qualifier	max_num_val
Number of qualifiers	max_qual
Number of rules in backward chain	max_rule_in_use
Number of variables	max_var

## Functions to Set Data

Set EXSYS Data	set_data
Assign a value at the start of a run	set_user_data

## Functions to Convert Names

Qualifier name to number	qual_name2num
Rule name to number	rule_name2num
Variable name to number	var_name2num

## Functions to Control GUI Environment

<i>Close EXSYS windows</i>	<i>close_active</i>
Display an EXSYS rule	display_rule
<i>Display a hypertext screen</i>	<i>exsys_hypertext</i>
Run the rules	exsys_runit
Set the name of the system	exsys_set_filename
<i>Display trace message</i>	<i>exsys_trace_print</i>
<i>Modify termination routine</i>	<i>exsys_xvt_terminate</i>
Display a file	file_display <i>file_display_nowait</i>
<i>Call EXSYS Notebook</i>	<i>notebook</i>
<i>Run a command file</i>	<i>run_cmd_file</i>

Run a report file

run\_report\_file

*Display a custom screen*

scrn\_display  
scrn\_display\_nowait

### **Functions to Handle Errors**

*Add an error message*

add\_to\_err\_msg

*Handle error message events*

exsys\_err\_msg\_timer

### **Functions to Handle Events in task\_eh**

*Handle E\_CHAR events*

do\_exsys\_task\_char

*Handle E\_CONTROL events*

do\_exsys\_task\_control

### **Utility Functions**

Drop CC- from text string

droprun

Convert a file to Stream LF

makeslf

Remove quotes

noquote

Test a file for Stream LF

slftest

Replace [[ ]] in string

varparse

## **add\_to\_err\_msg( )**

Add an error message to the list to be displayed based on a timer

### **Syntax:**

```
add_to_err_msg( s)
               char *s;                      /* the error message */
```

It is convenient to display most error messages with the `xvt_error( )` command. However, XVT does not allow this command to be called during `E_UPDATE` events. Certain types of errors are only detected during the `E_UPDATE` event. To allow appropriate error messages to be displayed, these messages are added to a list and displayed based on a timer. The `exsys_err_msg_timer( )` function displays any messages in the list. The call to this function is found in the `task_eh` section of `ex_link2.c`.

### **Use:**

This function can be called to add a message to the error list. It should only be used if directly calling the `xvt_error( )` command during an `E_UPDATE` event is unavoidable. (XVT does not allow calling `xvt_error( )` during `E_UPDATE`). In that case, use `add_to_err_msg( )` instead.

### **Example:**

We have an `E_UPDATE` event that can detect an error, we could use this function to display the error. Using an `xvt_error( )` command at this point would produce an error from XVT.

```
case E_UPDATE:
    .
    .
    .
    if (found_an_error == TRUE)
        add_to_err_msg("Error found at X");
```



## **choice\_text( )**

Get the text associated with a choice

### **Syntax:**

```
char  *choice_text(n)
      int n;           /* number of the choice */
```

This function returns the text associated with a choice. The function is passed the number of the choice.

After calling this function, either copy the string to a buffer, or use it immediately. The pointer returned is not guaranteed to have the data at a later time.

### **Use:**

This function can be used when creating screens that present data on the status of the choices.

**Note:** Do not use the earlier non-GUI form of this command, `get_choice( )`. Unfortunately `get_choice( )` was an internal XVT routine, and in earlier versions of XVT, it will link correctly - however it will not run.

### **Example:**

Print a list of the status of all choices:

```
for (i=1; i<=max_choice( ); i++) /* for each choice */
    fprintf(f, "%s : %lf\n", choice_text(i),
            get_choice_value(i));
```

## **close\_active( )**

Close all active EXSYS Runtime Windows

### **Syntax:**

```
close_active( )
```

This function closes all active EXSYS windows that may be asking a question or displaying data during a run. The top level window `exsys_win` is NOT closed. Normally this command should only be used in applications where the EXSYS inference engine is embedded within another application.

### **Use:**

This function is used when another application has the EXSYS inference engine embedded and needs to call EXSYS multiple times. Calling the `close_active( )` function will close all active windows. Then calling the XVT command to hide the `exsys_win` will make EXSYS effectively disappear. EXSYS can be restarted with the XVT command to show the EXSYS window, `exsys_win`, followed by the `runit( )` command.

### **Example:**

In the demo of embedding EXSYS, we created a custom command named `cc-close_exsys` with the associated function:

```
cmd0(argc, argv)
    int argc;
    char **argv;
{
    close_active( );
    /* close all active EXSYS windows */
    process_events( );
    /* make sure they are finished closing */
    show_window(exsys_win, FALSE);
    /* hide exsys_win */
}
```

When this command is called, the EXSYS windows will be closed and hidden.

Adding a command file to the knowledge base allows us to call this custom command at the end of each run:

```
rules all
results
cc-close_exsys( )
```

## **display\_rule( )**

Display a rule in an EXSYS window

### **Syntax:**

```
void display_rule(n)
    int n;                /* rule number */
```

This function displays a rule in the normal EXSYS rule display window. All rule display options to ask about data, examine notes or references, etc. are active.

### **Use:**

An application may wish to customize the way questions are asked or data is presented. The `display_rule( )` function allows customized interfaces to easily call the high level EXSYS functions to display a rule. All of the rule display options which allow interrogation on how data was set, what data is known and the source of data are available.

This function allows a high level rule display option to be added to custom routines with minimum effort.

### **Example:**

If we have a rule named "COST" and we want to display it:

```
rnum = rule_name2num("COST");
if (rnum != -1)    /* if good value */
    display_rule(rnum); /* display the rule */
```

## do\_exsys\_task\_char( )

Handle E\_CHAR events in task\_eh

### Syntax:

```
do_exsys_task_char( )
```

This function handles events for character key strokes that EXSYS accepts in task\_eh. For the EXSYS Runtime, these are only the ENTER (RETURN) keys to dismiss screens. The call to the do\_exsys\_task\_char( ) function is found in ex\_link2.c.

### Use:

This function should normally be left in the E\_CHAR part of task\_eh in ex\_link2.c. If EXSYS is embedded and an alternate task\_eh is created, it is the developers responsibility to determine which events are for EXSYS in task\_eh and which events are for developer created controls. All E\_CHAR events to EXSYS in task\_eh should be passed to do\_exsys\_task\_char( ). Events in task\_eh that are handled by the developer should NOT be passed to do\_exsys\_task\_char( ).

### Example:

In ex\_link2.c the task\_eh( ) callback function handles E\_CHAR events with:

```
case E_CHAR:
    /* EXSYS only uses one button in TASK_WIN.          */
    /* This is the "Run Expert System" button.          */
    /* EXSYS accepts the RETURN (ENTER) key as          */
    /* equivalent to pressing the button. Since          */
    /* there are no other controls, EXSYS's handling    */
    /* of E_CHAR events in TASK_WIN is simplified.      */
    /* If the program uses other E_CHAR events          */
    /* in TASK_WIN, add code to recognize and handle    */
    /* the other E_CHAR events and pass only the        */
    /* ENTER to EXSYS. If this is difficult, or          */
    /* impossible, delete do_exsys_task_char( ) below   */
    /* and handle all TASK_WIN E_CHAR events as for     */
    /* the user functions. The only EXSYS function     */
    /* that will be lost is the user will not be able  */
    /* to just hit ENTER and have the system start.    */
    /* Clicking on the button will still work.          */
    do_exsys_task_char(ep->v.chr.ch);
    return;
```

## do\_exsys\_task\_control( )

Handle E\_CONTROL events in task\_eh

### Syntax:

```
do_exsys_task_control( )
```

This function handles events for the normal controls that EXSYS draws in task\_eh. For the EXSYS Runtime, these are the buttons to start the expert system and dismiss the starting or ending text. The call to the do\_exsys\_task\_control( ) function is found in ex\_link2.c.

### Use:

This function should normally be left in the E\_CONTROL part of task\_eh in ex\_link2.c. If EXSYS is embedded and an alternate task\_eh is created, it is the developers responsibility to determine which events are for EXSYS controls in task\_eh and which events are for developer created controls. All events to EXSYS controls in task\_eh should be passed to do\_exsys\_task\_control( ). Events for controls in task\_eh that are created by the developer should NOT be passed to do\_exsys\_task\_control( ).

### Example:

In ex\_link2.c the task\_eh( ) callback function handles E\_CONTROL events with:

```
case E_CONTROL:
    /* EXSYS only uses one button in TASK_WIN. */
    /* This is the "Run Expert System" button. Since */
    /* there are no other controls, EXSYS's handling */
    /* of E_CONTROL events in TASK_WIN is simplified. */
    /* If the program creates and uses other controls */
    /* in TASK_WIN, add code to recognize which */
    /* events come from user controls. If the event */
    /* is from a user created control, add code to */
    /* handle the event. If the control is an EXSYS */
    /* control, call do_exsys_task_control( ). */
    /* DO NOT call do_exsys_task_control( ) for events */
    /* from user controls. */

    do_exsys_task_control( );
    return;
```

## droprun( )

Delete custom command text from prompt

### Syntax:

```
droprun(s )  
    char *s;      /* prompt string */
```

When a qualifier or variable has an associated custom command, the text of the custom command is part of the prompt string. (The prompt string being the text of a variable or a qualifier.) In some cases only the actual prompt text is wanted. The `droprun( )` command will take a string with a `CC-...( )` before the prompt and delete the custom command. The `droprun( )` command will also work for all other EXSYS internal commands that might precede the text of a prompt.

### Use:

The `droprun( )` command would be used within a custom command that displayed the prompt of a qualifier or variable without the associated `CC_..( )` or other EXSYS internal command.

### Example:

We have a custom command named `CC-GETDTA` and it was associated with a variable that has the prompt "The number of points". The actual prompt for the variable will be:

```
CC-GETDATA(1234) The number of points
```

If this string is passed to `droprun( )`, we would get:

```
The number of points
```

In a custom command we might use:

```
strcpy(b, get_var(5));  
droprun(b);  
win_draw_text(win, 10, 10, b, -1);
```

## **exsys\_err\_msg\_timer( )**

Display error messages based on a timer

### **Syntax:**

```
exsys_err_msg_timer( )
```

This function displays certain types of error messages. EXSYS displays most error messages with the `xvt_error( )` command. XVT does not allow this command to be called during `E_UPDATE` events. Unfortunately in dynamic custom screens, certain types of errors are only detected during the `E_UPDATE` event. To allow appropriate error messages to be displayed, these messages are added to a list and displayed based on a timer. The `exsys_err_msg_timer( )` function displays any messages in the list. The call to this function is found in `ex_link2.c`.

### **Use:**

This function should normally be left in the `E_TIMER` part of `task_eh` in `ex_link2.c`. If EXSYS is embedded and an alternate `task_eh` is created, add this function to handle `E_TIMER` events. Error messages are added to the list to be displayed with the `add_to_err_msg( )` function.

### **Example:**

In `ex_link2.c` the `task_eh( )` callback function handles `E_TIMER` events with:

```
case E_TIMER:
    /* Due to restrictions in XVT on displaying          */
    /* xvt_error( ) during E_UPDATE events, EXSYS        */
    /* displays some error messages on a timer. This     */
    /* timer is set in exsys_init_err_msg( ) above.      */
    /* The error messages may be set during the run.    */
    /* To display the errors, call                       */
    /* the exsys_err_msg_timer( ) function.              */
    exsys_err_msg_timer( );
    break;
```



## **exsys\_hypertext( )**

Call the EXSYS hypertext function

### **Syntax:**

```
exsys_hypertext( s)
char    *s;          /* the hypertext key word */
```

This function allows the EXSYS hypertext function to be called to display a screen. The associated screen should be in the .scr file and be defined like any other EXSYS hypertext screen. (See the EXSYS manual for details on the EXSYS hypertext system.)

The word passed to the function is the hypertext keyword. The screen associated with this word will be displayed. (Note: the hypertext screen identifier will be designated ~~ word)

### **Use:**

This function is a convenient way to enable the end user to make custom commands that access the hypertext screen. Remember that in most operating systems, the window of a hypertext screen can not be displayed on top of a modal dialog. If your custom screens are based on modal dialogs, they will not be able to bring up hypertext screens.

### **Example:**

A custom command that has a notebook button which allows the end user to make notes in a specific file:

```
case HYPertext_BTN:
    exsys_hypertext("widget");
    break;
```

## exsys\_runit( )

Restart an EXSYS application

### Syntax:

```
exsys_runit( )
```

This function restarts an already loaded EXSYS application. It will reinitialize all data (losing data from any previous run) and start at the beginning of any associated command file.

### Use:

This function should **ONLY** be used for embedded systems where EXSYS will be started and stopped multiple times. In all other cases, use the default running of rules or a command file which will load and run the rules following the creation of exsys\_win.

Normally exsys\_runit( ) would only be used after a redisplay of the exsys\_win, or in response to a request to restart the application from the beginning. **Use of exsys\_runit( ) in other cases will probably crash your application.**

### Example:

In the demo of embedding EXSYS, we handle the button to run EXSYS by creating exsys\_win the first time, and calling exsys\_runit( ) on subsequent calls:

```
case EXSYS_BTN:
    if (exsys_win == (WINDOW) NULL)
    {
        /* First time, so create the window */
        set_rect( ...);

        exsys_win = create_window(...);

        exsys_init(win);
        process_events( );
        exsys_fileopen( );
    }
    else /* already created, just hidden */
    {
        show_window(exsys_win, TRUE);
        exsys_runit( );
    }

    break;
```

## **exsys\_set\_filename( )**

Set the name of the expert system to be run

### **Syntax:**

```
exsys_set_filename(f)
    file *f;                                /* file name */
```

This function sets the name of the knowledge base to run. The file name passed only needs to be the name up to the period, not the file extension. (For example, a knowledge base may have files kb.rul and kb.txt. Only the "kb" part needs to be used.)

### **Use:**

This function is used ONLY when the knowledge base files are read from the disk. It is NOT used if the knowledge base has been converted to C code. The function must be used PRIOR to calling the create\_window( ) command for exsys\_win. If there is only a single knowledge base that will be used, it may be best to call this in main ( ) following the exsys\_set\_cfg( ) function. If this command is not used, when the expert system starts, it will wait for the user to select a file with the File Open menu option.

### **Example:**

In an application that will use the knowledge base EMBDEMO which will be read from disk:

```
main(int argc, char *argv[])
{
    static XVT_CONFIG config;

    exsys_set_cfg(argv[0]);

    exsys_set_filename("embdemo");

    for (i=1; i<argc; i++)
        exsys_set_param(argv[i]);

    config.appl_name = "EXSYSP";
    config.taskwin_title = "EXSYS Pro Runtime";
```

## **exsys\_trace\_print( )**

Display a message in the EXSYS trace file and window

### **Syntax:**

```
exsys_trace_print(s )  
    char *s;          /* message string */
```

This function displays a message in the EXSYS trace window (if that is open) and in the trace file (if one has been specified). The function automatically checks for the existence of the trace window and file. If they have not been created, no action will be taken.

### **Use:**

If your application has custom commands which obtain or manipulate data, you may wish to put messages in the trace window, especially when debugging your application. This function makes it easy to do this. The messages will appear in the trace window and trace file.

### **Example:**

To add a trace statement in a custom command:

```
cmd23(argc, argv)  
    int argc;  
    char **argv;  
    {  
        char b[100];  
        char retdata[100];  
        .  
        .  
        sprintf(b, "cmd23 obtained %s", retdata);  
        exsys_trace_print(b);  
        .  
        .  
    }
```

## **exsys\_xvt\_terminate( )**

User modifiable routine called whenever EXSYS terminates

### **Syntax:**

```
exsys_xvt_terminate( )
```

This function is defined in `ex_link.c`. Normally it is defined to just call `xvt_terminate( )` which will terminate the application. Since this module is provided as part of the source code in Linkable Objects, the end user can modify the function of `exsys_xvt_terminate( )`.

### **Use:**

The `exsys_xvt_terminate( )` function may be modified to perform "clean up" functions before terminating the application such as deallocating memory or closing open files. By placing these functions in `exsys_xvt_terminate( )`, it is guaranteed that any exit from EXSYS will perform the desired clean up. Also, since this routine can be called from the end user's code, it can also be used to terminate the application outside of the EXSYS portion.

In some cases, such as the embedded system demo. It may be desirable to have the end of the expert system NOT terminate the application. In this case, `exsys_xvt_terminate( )` can be modified to not perform any function - just return.

### **Example:**

In an application that opens a file which must be closed before the application terminates, `exsys_xvt_terminate( )` in `ex_link.c` could be modified to:

```
exsys_xvt_terminate( )
{
    fclose(myfile);
    xvt_terminate( );
}
```

## **file\_display( )**

Display a text file in a window and wait

### **Syntax:**

```
file_display(f)
    file *f;                                /* file name */
```

This function displays a TEXT file in a window. The end user can scroll the text within the window. EXSYS automatically handles all events for the window. If there are any hypertext words in the displayed text, they will be highlighted automatically. A double click on a hypertext word will display the associated hypertext custom screen.

The program will wait for the user to dismiss the text display window by clicking on OK before it will return from the `file_display( )` command.

### **Use:**

This commands provides a high level command to easily display a file of text. This function can be used to display reports produced by the system, or to display files of other information that the end user may wish to see.

To have a window display text and the program continue without waiting for the user to dismiss the window, use the `file_display_nowait( )` command.

### **Example:**

Display a file of help information when a button is clicked. The callback function associated with the window containing the button would have a section of code that would be used if the button was clicked:

```
case HELP_BUTTON:
    file_display("help1.dat");
    break;
```

## **file\_display\_nowait( )**

Display a text file in a window and don't wait

### **Syntax:**

```
file_display_nowait(f)
    file *f;                                /* file name */
```

This function displays a TEXT file in a window. The end user can scroll the text within the window. EXSYS automatically handles all events for the window. If there are any hypertext words in the displayed text, they will be highlighted automatically. A double click on a hypertext word will display the associated hypertext custom screen.

The program will NOT wait for the user to dismiss the text display window and will immediately return from the file\_display\_nowait( ) command.

### **Use:**

This command provides a high level command to easily display a file of text. This function can be used to display reports produced by the system, or to display files of other information that the end user may wish to see.

To have a window display text and wait before it continues, use the file\_display( ) command.

### **Example:**

Display two files of help information when a button is clicked. Both will be displayed at the same time and the user can examine both together. Since the file\_display\_nowait( ) command returns immediately, both text windows will be displayed. (If the file\_display( ) command had been used instead, the first window would be displayed and only after it was closed would the second window be displayed.) The callback function associated with the window containing the button would have a section of code that would be used if the button was clicked:

```
case HELP_BUTTON:
    file_display_nowait("help1.dat");
    file_display_nowait("help2.dat");
    break;
```

## **get\_choice\_value( )**

Get the value of a choice

### **Syntax:**

```
double get_choice_value(n)
    int n;                /* number of the choice */
```

This function returns the value of a choice. The function is passed the number of the choice.

### **Use:**

This function can be used when creating screens that present data on the status of the choices.

### **Example:**

Print a list of the status of all choices:

```
for (i=1; i<=max_choice( ); i++) /* for each choice */
    fprintf(f, "%s : %lf\n", choice_text(i),
            get_choice_value(i));
```



## **get\_Nvar\_value( )**

Get the value of a numeric variable

### **Syntax:**

```
double get_Nvar_value(n)
    int n;                /* number of the variable */
```

This function returns the value of a numeric variable as a double precision float. The function is passed the number of the variable.

### **Use:**

This function can be used when creating screens that present data on the status of the variables. To check if a variable is string, numeric or text, use the `var_type( )` function. Do not ask for the numeric value of a string variable.

### **Example:**

Print a list of the status of all variables that have values set:

```
for (i=1; i<=max_var( ); i++) /* for each variable */
{if (var_set(i) == TRUE)
    {if (var_type(i) == 'N') /* if numeric */
        fprintf(f, "%s : %lf\n", get_var(i),
                get_Nvar_value(i));
        else if (var_type(i) == 'S') /* if string */
            fprintf(f, "%s : %s\n", get_var(i),
                    get_Svar_value(i));
    }
}
```

## **get\_qual( )**

Get the text associated with a qualifier

### **Syntax:**

```
char *get_qual(n)
    int n;                /* number of the qualifier */
```

This function returns the qualifier text associated with qualifier number *n*. ONLY the text of the qualifier is returned, no value text is returned.

After calling this function, either copy the string to a buffer, or use it immediately. The pointer returned is not guaranteed to have the data at a later time.

### **Use:**

This function can be used to get text for use in a screen to ask a question. If the current state of a qualifier is needed, the `get_qv_set( )` function should be used. If the text of a qualifier's value is needed, the `get_qval( )` function should be used.

If the text of the qualifier starts with a CC-.. or other internal EXSYS command, and you wish to have this command text stripped off the qualifier, call the `droprun( )` command.

### **Example:**

In a window that asks the end user to select a value for the qualifier, there would be an `E_UPDATE` section for the window. This code prints "Please select a value for" and then prints the text of the qualifier below it. (This would then have value mouse regions or buttons below this.)

```
win_draw_text(win,10,30,"Please select a value for",-1);
strcpy(b, get_qual(n));    /* Get the text string */
win_draw_text(win, 10, 60, b, -1);
```

## **get\_qv\_set( )**

Get the text associated with a qualifier and all values that have been set

### **Syntax:**

```
char *get_qv_set(n)
    int n;                /* number of the qualifier */
```

This function returns the text associated with a qualifier concatenated with all of the values that have been set.

The function is passed the number of the qualifier.

After calling this function, either copy the string to a buffer, or use it immediately. The pointer returned is not guaranteed to have the data at a later time.

### **Use:**

This function can be used in reports to display the current state of the data - such as the Known Data window. If only the text of a qualifier is needed, the `get_qual( )` function should be used. If the text of the qualifier's values is needed, the `get_qval( )` function should be used.

If the text of the qualifier starts with a CC-.. or other internal EXSYS commands, and you wish to have this command text stripped off the qualifier, call the `droprun( )` command.

### **Example:**

Print all of the qualifiers that have had a value set:

```
for (i=1; i<=max_qual( ); i++) /* for each qualifier */
    {if (qual_set(i) == TRUE) /* if it was set */
        fprintf(f, "%s\n", get_qv_set(i));
    }
```

## get\_qval( )

Get the text associated with a qualifier's value

### Syntax:

```
char *get_qval(n, v)
    int n;          /* number of the qualifier */
    int v;          /* number of the value */
```

This function returns the text associated with a qualifier's values. The function is passed the number of the qualifier and the number of the specific value. The number of the values starts with 1.

After calling this function, either copy the string to a buffer, or use it immediately. The pointer returned is not guaranteed to have the data at a later time.

### Use:

This function can be used to get text for use in a screen to ask a question. If the current state of a qualifier is needed, the `get_qv_set( )` function should be used. If the text of only the qualifier is needed, the `get_qual( )` function should be used.

### Example:

In a window that asks the end user to select a value for the qualifier, there would be an `E_CREATE` section for the window. This code creates a set of buttons for each of the possible values:

```
for (i=1; i<=max_num_val(n); i++)
{set_rect(&rct, 10, 10+20*i, 150, 30+20*i);
  val_button[i] = create_control(WC_PUSHBUTTON,
                                &rct,
                                get_qval(n, i),
                                win,
                                CTL_FLAG_DEFAULT,
                                01,
                                i);
}
```

## **get\_qval\_value( )**

Test if a particular value in a qualifier has been set

### **Syntax:**

```
BOOLEAN get_qval_value(n, v)
    int n;          /* number of the qualifier */
    int v;          /* number of the value */
```

This function returns tests to see if a particular value in a qualifier has been set. The function is passed the number of the qualifier and the number of the value. Value numbers start at 1.

### **Use:**

This function can be used when creating screens that present data on the status of the qualifiers.

### **Example:**

Print a list of the values set for a qualifier:

```
for (i=1; i<=max_num_val(n); i++) /* for each choice */
    {if (get_qval_value(n, i) == TRUE)
        fprintf(f, "%s\n", get_qval(n, i));
    }
```

## **get\_Svar\_value( )**

Get the value of a string variable

### **Syntax:**

```
char    *get_Svar_value(n)
      int n;                /* number of the variable */
```

This function returns the value of a of a string variable as a character string pointer. The function is passed the number of the variable.

After calling this function, either copy the string to a buffer, or use it immediately. The pointer returned is not guaranteed to have the data at a later time.

### **Use:**

This function can be used when creating screens that present data on the status of the variables. To check if a variable is string, numeric or text, use the `var_type( )` function. Do not ask for the string value of a numeric variable.

### **Example:**

Print a list of the status of all variables that have values set:

```
for (i=1; i<=max_var( ); i++) /* for each choice */
{if (var_set(i) == TRUE)
    {if (var_type(i) == 'N') /* if numeric */
        fprintf(f, "%s : %lf\n", get_var(i),
                get_Nvar_value(i));
        else if (var_type(i) == 'S') /* if string */
            fprintf(f, "%s : %s\n", get_var(i),
                    get_Svar_value(i));
    }
}
```

## **get\_var( )**

Get the text associated with a variable

### **Syntax:**

```
char  *get_var(n)
      int n;           /* number of the variable */
```

This function returns the prompt text associated with a variable. The function is passed the number of the variable.

After calling this function, either copy the string to a buffer, or use it immediately. The pointer returned is not guaranteed to have the data at a later time.

### **Use:**

This function can be used when creating screens that ask the user for input on a specific variable.

If the text of the prompt starts with a CC-.. or other internal EXSYS command, and you wish to have this command text stripped off the prompt, call the `droprun( )` command.

### **Example:**

In a window that asks the end user to select a value for the variable, there would be an `E_UPDATE` section for the window. This code prints "Please input a value for" and then prints the text of the variable below it. (This would then have an edit region below this.)

```
win_draw_text(win,
              10,
              30,
              "Please input a value for",
              -1);

strcpy(b, get_var(n)); /* Get the text string */
win_draw_text(win, 10, 60, b, -1);
```

## **makeslf( ) (VMS Only)**

Convert a file to Stream LF type

### **Syntax:**

```
makeslf(s )  
    char *s;      /* name of file */
```

Many EXSYS operations require random repositioning to read within a file. In VMS this operation can only be done on Stream LF files. This function converts a file to type Stream LF. This applies only to VMS. The type of a file can be obtained with the EXSYS API function `slftest( )`.

### **Use:**

If your application is creating or modifying files in VMS that EXSYS needs to handle as Stream LF, call this function to convert them.

This function is never needed for operating system other than VMS.

Note: In most cases, this should NOT be necessary since EXSYS will automatically check all files that it needs to be Stream LF and convert them.

### **Example:**

To check a file and convert it:

```
if (slftest("xxx") == 0)  
    makeslf("xxx");
```



## **max\_choice( )**

Number of choices in a system

### **Syntax:**

```
max_choice( )
```

This function returns the number of choices in a system.

### **Use:**

This function is typically used in a custom command that handles all the choices in a system. It can be used to have the command automatically adjust to changes in the number of choices.

### **Example:**

In a custom command that displays all choices:

```
for (i=1; i<=max_choice( ); i++)  
    fprintf(f, "%s : %lf\n", choice_text(i),  
            get_choice_value(i));
```

## **max\_num\_val( )**

Number of values associated with a qualifier

### **Syntax:**

```
max_num_val(n )  
    int n;                /* number of the qualifier */
```

Qualifiers can have up to 30 values associated with them. This function returns the number of values associated with a specific qualifier. If the qualifier number is out of range, an error message will be displayed.

### **Use:**

This function is typically used in a custom command that handles the qualifiers generically and needs to be able to adjust to the number of values in a particular qualifier.

### **Example:**

In a custom command that creates a button for each value of a qualifier:

```
for (i=1; i<=max_num_val(n); i++)  
{set_rect(&rct, 10, 10+20*i, 150, 30+20*i);  
  val_button[i] = create_control(WC_PUSHBUTTON,  
                                &rct,  
                                get_qval(n, i),  
                                win,  
                                CTL_FLAG_DEFAULT,  
                                01,  
                                i);  
}
```

## **max\_qual( )**

Number of qualifiers in a system

### **Syntax:**

```
max_qual( )
```

This function returns the number of qualifiers in a system.

### **Use:**

This function is typically used in a custom command that handles all the qualifiers in a system. It can be used to have the command automatically adjust to changes in the number of qualifiers.

### **Example:**

In a custom command that displays all qualifiers that have had a value set:

```
for (i=1; i<=max_qual( ); i++)
{if (qual_set(i) == TRUE)
    fprintf(f, "%s\n", get_qv_set(i));
}
```

## **max\_rule\_in\_use( )**

Number of rules in the backward chain

### **Syntax:**

```
max_rule_in_use( )
```

Backward chaining can result in one rule invoking another rule to derive needed data. This produces a list of rules that are currently being processed. When the user asks "WHY" to an EXSYS question, this list of rules is displayed. Custom interfaces can also obtain information on the list of rules currently being processed with the `rule_in_use( )` command.

The argument passed to the function is the number of the item in the list: `rule_in_use(1)` will return the number of the current rule being processed by the system, `rule_in_use(2)` will return the number of the rule which caused the current rule to be invoked, and `rule_in_use(3)` will return the number of the rule that cause the level 2 rule to be invoked, etc., until the last rule which is the lowest level rule being processed is reached. The number of the lowest level rule is obtained with the **max\_rule\_in\_use( )** function.

### **Use:**

This function is typically used with `rule_in_use( )` in a custom command that allows the end user to ask why a question is being asked. The `max_rule_in_use( )` function tells the user the largest valid value to pass to `rule_in_use( )`.

### **Example:**

A custom command that asks a question and allows the end user to ask WHY:

```
maxrule = max_rule_in_use( );
for (i=1; i<=maxrule( ); i++)
    /* for each rule in the list */
    display_rule(rule_in_use(i));
```

## **max\_var( )**

Number of variables in a system

### **Syntax:**

```
max_var( )
```

This function returns the number of variables in a system.

### **Use:**

This function is typically used in a custom command that handles all the variables in a system. It can be used to have the command automatically adjust to changes in the number of variables.

### **Example:**

In a custom command that displays all variables that have had a value set:

```
for (i=1; i<=max_var( ); i++)
{if (var_set(i) == TRUE)
    fprintf(f, "%s : %lf\n", get_var(i),
        get_Nvar_value(i));
}
```

## noquote( )

Remove the " " from a string

### Syntax:

```
noquote( s)
char    *s;      /* the string */
```

This function removes any " " that may start and end a string. If there are no " " the string is unchanged. NOTE: The string passed is directly modified. This function does NOT return a pointer to a modified string.

### Use:

This function is a convenient way to drop quotes. The arguments passed to a custom command may have quotes if the calling function used them to either indicate a string with spaces embedded or to designate the name of a variable that was to be passed as a string rather than as a value to be evaluated.

### Example:

A custom command is passed the name of the variable to assign data to:

```
CC-MYDATA(" [X] ", ....)
```

The [X] is put in quotes to pass it as a string. If there were no quotes the value of [X] would be determined and passed as a parameter.

Within the custom command, we want to build a string to assign data to the variable specified.

```
noquote(argv[0]);
sprintf(b, "%s %lf", argv[0], value_to_assign);
set_data(b);
```

We MUST call noquote( ) before we use the string because it will have quotes around it. This way we will build:

```
[X] 1234
```

Without the noquote( ) we would build:

```
"[X]" 1234
```

Which would not work in the set\_data( )

## **notebook( )**

Call the EXSYS Notebook function

### **Syntax:**

```
notebook( f)
        char    *f;          /* the name of the file */
```

This function allows the EXSYS Notebook function to be called to make short notes in a file. The notebook file can be written to or examined. The name passed to the function is the name of the notebook file to use.

### **Use:**

This function is a convenient way to enable the end user to make short comments about the execution of the expert system.

### **Example:**

A custom command that has a notebook button which allows the end user to make notes in a specific file:

```
case NOTEBOOK_BTN:
    notebook("myfile");
    break;
```



## **qual\_name2num( )**

Convert a qualifier name to the associated number

### **Syntax:**

```
int  qual_name2num(s)
      char *s;                /* qualifier name */
```

This function converts a qualifier name to the associated qualifier number. The function is passed the name of the qualifier and returns the number. If there is no qualifier that matches the name, a value of -1 is returned.

### **Use:**

Many of the EXSYS API functions use the number of a qualifier as a parameter. However, within a system, it may be easier to refer to qualifiers by a name that represents the meaning of the qualifier. Using names is recommended since number can easily change if a qualifier is deleted or reordered.

The purpose of the qual\_name2num( ) function is to allow names to be used and still convert them to the appropriate value for other API calls.

### **Example:**

If we have a qualifier named "COLOR" and we want to print its current status:

```
qnum = qual_name2num("COLOR");
if (qnum != -1)    /* if good value */
    fprintf(f, "%s", get_qv_set(qnum) );
```

## **qual\_set( )**

Determines if a qualifier has a value set

### **Syntax:**

```
qual_set( n)
      int  n;      /* the qualifier number */
```

This function returns TRUE if the qualifier had a value set by either the inference engine, an external source, or by asking the user for data. If no value was set, the function returns FALSE.

### **Use:**

This function is typically used in a custom command that handles all the qualifiers in a system, but which needs to select those that had a value set.

### **Example:**

In a custom command that displays all qualifiers that have had a value set:

```
for (i=1; i<=max_qual( ); i++)
{if (qual_set(i) == TRUE)
    fprintf(f, "%s\n", get_qv_set(i));
}
```

## **rule\_in\_use( )**

Rules currently active in the backward chain

### **Syntax:**

```
rule_in_use( n)
           int n;      /* the number in the list*/
```

Backward chaining can result in one rule invoking another rule to derive needed data. This produces a list of rules that are currently being processed. When the user asks "WHY" to an EXSYS question, this list of rules is displayed. Custom interfaces can also obtain information on the list of rules currently being processed with the `rule_in_use( )` command.

The argument passed to the function is the number of the item in the list: `rule_in_use(1)` will return the number of the current rule being processed by the system, `rule_in_use(2)` will return the number of the rule which caused the current rule to be invoked, and `rule_in_use(3)` will return the number of the rule that cause the level 2 rule to be invoked, etc., until the last rule which is the lowest level rule being processed is reached. The number of the lowest level rule is obtained with the `max_rule_in_use( )` function.

### **Use:**

This function is typically used in a custom command that allows the end user to ask why a question is being asked. The `rule_in_use( )` function allows the backward chain to be examined.

### **Example:**

A custom command that asks a question and allows the end user to ask WHY:

```
maxrule = max_rule_in_use( );
for (i=1; i<=maxrule( ); i++)
    /* for each rule in the list */
    display_rule(rule_in_use(i));
```

## **rule\_name2num( )**

Convert a rule name to the associated number

### **Syntax:**

```
int rule_name2num(s)
    char *s;                /* rule name */
```

This function converts a rule name to the associated rule number. The function is passed the name of the rule and returns the number. If there is no rule that matches the name, a value of -1 is returned.

### **Use:**

Some of the EXSYS API functions use the number of a rule as a parameter. However, within a system, it is easier to refer to rules by a name that represents the meaning of the rule. Using names is recommended since number can easily change if a rule is deleted or reordered.

The purpose of the rule\_name2num( ) function is to allow names to be used and still convert them to the appropriate value for other API calls.

## **run\_cmd\_file( )**

Execute the commands in a command file

### **Syntax:**

```
run_cmd_file(f)
           file *f;                      /* file name */
```

This function executes the commands in a command file. All of the commands that can be included in a .CMD file can be included.

### **Use:**

This command provides a very high level command to easily access a wide range of EXSYS functionality. This function can be used to make a system execute in different ways based on user choices. Several alternate command files can be defined to perform particular operations. The end user may select what operation to perform by clicking on a particular button. The command file may execute any of the EXSYS Command Language commands. This provides tremendous flexibility in having user selections change the operation of the system.

### **Example:**

Execute a series of commands when a particular button is clicked. The callback function associated with the window containing the button would have a section of code that would be used if the button was clicked:

```
case EXEC_1_BUTTON:
    run_cmd_file("set1cmds");
    break;
```

## **run\_report\_file( )**

Execute the commands in a report file

### **Syntax:**

```
run_report_file(f)
    file *f;                                /* file name */
```

This function executes the commands in a report file. All of the commands that can be included in a .OUT file can be used.

### **Use:**

This commands provides a very high level command to easily generate a wide range of reports. This function can be used to make a system produce different reports based on user choices. Several alternate report files can be defined to generate different types of output. The end user may select what report to perform by clicking on a particular button. The report file may execute any of the EXSYS Report Generator commands. This provides tremendous flexibility in having user selections change the operation of the system.

### **Example:**

Produce a report when a particular button is clicked. The callback function associated with the window containing the button would have a section of code that would be used if the button was clicked:

```
case REPORT_BUTTON:
    run_report_file("type1.rpt");
    break;
```

## **scrn\_display( )**

Display a custom screen definition file in a window and wait

### **Syntax:**

```
scrn_display(f)
      file *f;                                /* file name */
```

This function displays a custom screen definition file in a window. The end user can use all of the controls within the window. EXSYS automatically handles all events for the window. If there are any hypertext words in the displayed text, they will be highlighted automatically. A double click on a hypertext word will display the associated hypertext custom screen.

The program will wait for the user to dismiss the custom screen window by clicking on OK before it will return from the scrn\_display() command.

The screen file can contain any custom screen commands and may be designed with the EXSYS Screen Design Program ExDesign.

Custom screens displayed with this command do NOT return data, even if return data commands are included. To return data, associate a custom screen in the .SCR file with an EXSYS qualifier or variable.

### **Use:**

This commands provides a high level command to easily display a custom screen. This function can be used to display reports produced by the system, or to display other information that the end user may wish to see.

To have a window display a custom screen and the program continue without waiting for the user to dismiss the window, use the scrn\_display\_nowait( ) command.

**Example:**

Display a screen of help information when a button is clicked. The callback function associated with the window containing the button would have a section of code that would be used if the button was clicked:

```
case  HELP_BUTTON:
    scrn_display("help1.dat");
    break;
```



## **scrn\_display\_nowait( )**

Display a custom screen definition file in a window and do NOT wait

### **Syntax:**

```
scrn_display_nowait(f)
    file *f;                                /* file name */
```

This function displays a custom screen definition file in a window. The end user can use the controls within the window. EXSYS automatically handles all events for the window. If there are any hypertext words in the displayed text, they will be highlighted automatically. A double click on a hypertext word will display the associated hypertext custom screen.

The program will NOT wait for the user to dismiss the custom screen display window and will immediately return from the `scrn_display_nowait( )` command. The screen file can contain any custom screen commands and may be designed with the EXSYS Screen Design Program ExDesign.

Custom screens displayed with this command do NOT return data, even if return data commands are included. To return data, associate a custom screen in the .SCR file with an EXSYS qualifier or variable.

### **Use:**

This command provides a high level command to easily display a custom screen. This function can be used to display reports produced by the system, or to display other information that the end user may wish to see.

To have a window display a custom screen and wait before it continues, use the `scrn_display( )` command.

### Example:

Display two custom screens of help information when a button is clicked. Both will be displayed at the same time and the user can examine both together. Since the `scrn_display_nowait( )` command returns immediately, both text windows will be displayed. (If the `scrn_display( )` command had been used instead, the first window would be displayed and only after it was closed would the second window be displayed.) The callback function associated with the window containing the button would have a section of code that would be used if the button was clicked:

```
case  HELP_BUTTON:
    scrn_display_nowait("help1.dat");
    scrn_display_nowait("help2.dat");
    break;
```

## set\_data( )

Assign a value to a qualifier, variable or choice

### Syntax:

```
set_data(s)
char *s;      /* assignment string */
```

This command allows the assigning of data to EXSYS qualifiers, variables and choices.

String s has the same syntax as a line from a return.dat file:

For qualifiers:   **Q** <qual #> <value(s)>  
                  **Q** "name" <value(s)>

For variables:   **V** <var #> <value>  
                  [var name] <value>

For choices:     **C** <choice #> <value>  
                  **C** "text" <value >

<qual #>, <var #> or <choice #> - The number of the qualifier, variable or choice that the data is being passed back for.

"name" - The name associated with the qualifier.

"text" - A unique text sub-string in the choice.

<value> - The data to assign.

### Use:

This function can be used to assign data in EXSYS. This can be used when a user defined window asks the user for input or when a custom command accesses some external source for data and needs to assign it to an EXSYS variable.

The function does not return a value, however if an invalid assignment string is passed to the command, an error message will be displayed and no assignment will be made.

EXSYS will automatically assign string values to string variables.

The `set_data( )` command can be used in custom commands or called in the `set_user_data( )` function. (See the section on `set_user_data( )`).

### Example:

For example:

1. `set_data("V3 2.1")` means assign a value of 2.1 to variable 3
2. `set_data("[X] 2.1")` means assign a value of 2.1 to variable [X]
3. If qualifier #7 has four values:

Qualifier #7

THE COLOR IS

1. RED
2. BLUE
3. GREEN
4. ORANGE

`set_data("Q7 1,2")` means for qualifier 7, set values 1 and 2 as true, or "The color is red or blue". If more than one value is set, the values must be separated by a space or comma.

If this qualifier has the associated name "COLOR", we could use

`set_data("Q 'COLOR' 1,2")`

4. We have a function `get_some_data( )` that returns a value from an external source, based on a string we pass to it. We call this function in a custom command that is passed a string to use in `get_some_data( )` and the name of the variable to assign the value to. The syntax of the custom command is:

`CC-DATA("search string" "[variable name]")`

The associated custom command would be:

```
double get_some_data(char *);
        /* a user defined function to get data */

cmd0(int argc, char **argv)
{
    char b[100];

    noquote(argv[1]);
    sprintf(b, "%s %lf",    /* build a string */
            argv[1],        /* the variable to assign to */
            get_some_data(argv[0])); /* the data */
    set_data(b);           /* assign it */
}
```

## **set\_user\_data( )**

Assign values at the start of a run

### **Syntax:**

Called automatically by EXSYS, NOT a user callable routine.

This function can contain `set_data( )` commands that are called automatically at the start of a run without using custom commands.

### **Use:**

The function `set_user_data( )` is defined in the module `ex_link.c`. The default version of the function just returns, but can be modified to assign data or perform other actions needed in the system.

This function can be used to assign data to EXSYS variables at the start of a run. This can be used similar to a `DATALIST` command or can be used to initialize particular variables. The `set_user_data( )` function may contain other C code and other EXSYS API commands. By the time EXSYS calls `set_user_data( )`, memory for all EXSYS data structures will have been allocated.

### **Example:**

If the `set_user_data( )` function in `ex_link.c` was modified to the following,

```
set_user_data( )
{
    set_data("Q1 2");
    set_data("[X] 5");
}
```

it would set qualifier 1 value 2 as TRUE and would assign a value of 5 to variable [X].

## **slftest( )    (VMS Only)**

Test if a file is a Stream LF file

### **Syntax:**

```
slftest(s )  
    char *s;      /* name of file */
```

Many EXSYS operations require random repositioning to read within a file. In VMS this operation can only be done on Stream LF files. This function tests if a file is of type Stream LF. This applies only to VMS. If a file is not Stream LF, it can be converted with the EXSYS API function `makeslf( )`.

This function returns 1 if the file is a Stream LF type and 0 if it is not.

### **Use:**

If your application is creating or modifying files in VMS that EXSYS needs to handle as Stream LF, call this function and `makeslf( )` to convert them.

This function is never needed for operating system other than VMS.

Note: In most cases, this should NOT be necessary since EXSYS will automatically check all files that it needs to have be Stream LF and convert them.

### **Example:**

To check a file and convert it:

```
if (slftest("xxx") == 0)  
    makeslf("xxx");
```

## **var\_name2num( )**

Convert a variable name to the associated number

### **Syntax:**

```
int  var_name2num(s)
      char *s;                /* variable name */
```

This function converts a variable name to the associated variable number. The function is passed the name of the variable and returns the number. If there is no variable that matches the name, a value of -1 is returned.

The variable name string can either just be the name or can be the name in [ ].

### **Use:**

Many of the EXSYS API functions use the number of a variable as a parameter. However, within a system, it is easier to refer to variables by a name that represents the meaning of the variable. Using names is recommended since number can easily change if a variable is deleted or reordered.

The purpose of the var\_name2num( ) function is to allow names to be used and still convert them to the appropriate value for other API calls.

### **Example:**

If we have a variable named "SIZE" and we want to print its current value:

```
vnum = var_name2num("SIZE");
if (vnum != -1)    /* if good value */
    fprintf(f, "%lf", get_Nvar_value(vnum) );
```



## **var\_set( )**

Determines if a variable has a value

### **Syntax:**

```
var_set( n)
      int  n;      /* the variable number */
```

This function returns TRUE if the variable had a value set by either the inference engine, an external source, or by asking the user for data. If no value was set, the function returns FALSE.

### **Use:**

This function is typically used in a custom command that handles all the variables in a system, but which needs to select those that had a value set.

### **Example:**

In a custom command that displays all variables that have had a value set:

```
for (i=1; i<=max_var( ); i++)
{if (var_set(i) == TRUE)
    fprintf(f, "%s : %lf\n", get_var(i),
                                get_Nvar_value(i));
}
```

## **var\_type( )**

Determines the type of a variable

### **Syntax:**

```
var_type( n)
        int  n;      /* the variable number */
```

This function determines the type of a variable. EXSYS variables can be numeric, string or text only. (See the EXSYS manual for details on the EXSYS variable types.)

This function will return:

'N'	for Numeric variables
'S'	for String variables
'T'	for Text Only variables

### **Use:**

This function is typically used in a custom command that handles all the variables in a system, but which needs to handle the various types differently.

### **Example:**

In a custom command that displays all variables that have had a value set:

```
for (i=1; i<=max_var( ); i++) /* for each variable */
{if (var_set(i) == TRUE)
    {vtype = var_type(i);

        if (vtype == 'N') /* if numeric */
            fprintf(f, "%s : %lf\n", get_var(i),
                    get_Nvar_value(i));
        else if (vtype == 'S') /* if string */
            fprintf(f, "%s : %s\n", get_var(i),
                    get_Svar_value(i));
    }
}
```

## **varparse( )**

Replace ([ ]) and <? ?> stings

### **Syntax:**

```
varparse(s )  
    char *s;      /* string */
```

EXSYS supports replaceable parameters in text string. If a variable is enclosed in [ ], the value of the variable will be placed in the string. Likewise, an expression in <? ?> will be evaluated and placed in the string. See the EXSYS manual for details on using [ ] and <? ?> replacements.

The varparse( ) function makes these replacements. If the value of a variable is not fully known, varparse( ) will invoke the EXSYS inference engine to determine a value, or may ask the end user for the value.

After varparse( ) returns, the string with all replacements will be in the original buffer passed to varparse( ). The varparse( ) function should ONLY be passed text in a buffer - never static data.

When a string buffer is passed to varparse( ), make sure that the buffer is large enough to handle the increased size of the string after replacements have been made.

### **Use:**

This function is typically used to handle embedded data that may be present in qualifiers, variables or choices that are displayed in a custom window.

### **Example:**

In a custom command that displays choices which may have embedded data, we might use:

```
strcpy(b, choice_text(5));  
varparse(b);  
win_draw_text(win, 10, 10, b, -1);
```

## **was\_rule\_used( )**

Test if a rule was used during a run

### **Syntax:**

```
Boolean was_rule_used(n)
      int  n;                /* rule number */
```

This function tests if a rule was used during a run. The rule may have been found to be TRUE or, if it had an ELSE part, found to be FALSE but still used. The function returns TRUE or FALSE.

### **Use:**

This function can be used to test which rules fired during a run.

### **Example:**

To display all rules that fired during a run:

```
for (i=1; i<= max_rule( ); i++)
    {if (was_rule_used(i) == TRUE)
        display_rule(i);
    }
```

# DISTRIBUTION LIST

AUL/LSE 1 cy  
Bldg. 1405 - 600 Chennault Circle  
Maxwell AFB, AL 36112-6424

DTIC/OCF 2 cys  
8725 John J. Kingman Rd Ste 944  
FT Belvoir, VA 22060-6218

AFSAA/SAI 1 cy  
1580 Air Force Pentagon  
Washington, DC 20330-1580

PL/SUL 2 cys  
Kirtland AFB, NM 87117-5776

PL/HO 1 cy  
Kirtland AFB, NM 87117-5776

Official Record Copy

PL/VTQ/ Capt Mary Boom 2 cys

Dr. R.V. Wick  
PL/VT 1 cy  
Kirtland AFB, NM 87117-5776



DEPARTMENT OF THE AIR FORCE  
PHILLIPS LABORATORY (AFMC)

28 Jul 97

MEMORANDUM FOR DTIC/OCP

8725 John J. Kingman Rd, Suite 0944  
Ft Belvoir, VA 22060-6218

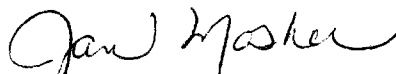
FROM: Phillips Laboratory/CA  
3550 Aberdeen Ave SE  
Kirtland AFB, NM 87117-5776

SUBJECT: Public Releasable Abstracts

1. The following technical report **abstracts** have been cleared by Public Affairs for unlimited distribution:

PL-TR-96-1020	ADB208308	PL 97-0318 (clearance number)
PL-TR-95-1093	ADB206370	PL 97-0317
PL-TR-96-1182	ADB222940	PL 97-0394 and DTL-P-97-142
PL-TR-97-1014	ADB222178	PL 97-0300

2. Any questions should be referred to Jan Mosher at DSN 246-1328.

  
Jan Mosher  
PL/CA

cc:  
PL/TL/DTIC (M Putnam)